The Compact Muon Solenoid Experiment
TriDAS Trigger and Data Acquisition

RU Builder
User Manual

Version 3.9
May 8, 2006

Authors: S. Murray
CI identifier EVB_D_18306

Fermilab

Revision History

Date	Version	Description	Author
July 11, 2003	0.1	Document creation	S. Murray
July 31, 2003	1.0	Finalization of version 1.0	J. Gutleber
May 24, 2004	2.0	Updated for version 2.0 of the EVB, now referred to as the RU builder	S. Murray
November 1, 2004	2.1	Updated for version 2.1 of the RU builder	S. Murray
May 2, 2005	3.0	Updated for version 3.0 of the RU builder	S. Murray
July 4, 2005	3.2	Updated for version 3.2 of the RU builder	S. Murray
August 16, 2005	3.3	Updated for version 3.3 of the RU builder	S. Murray
August 26, 2005	3.4	Updated for version 3.4 of the RU builder	S. Murray
October 5, 2005	3.5	Updated for version 3.5 of the RU builder	S. Murray
October 27, 2005	3.6	Updated for version 3.6 of the RU builder	S. Murray
January 30, 2006	3.7.1	Updated for version 3.7.1 of the RU builder	S. Murray
March 29, 2006	3.8	Updated for version 3.8 of the RU builder	S. Murray
May 8, 2006	3.9	Updated for version 3.9 of the RU builder	S. Murray

CI Record

Field	Description
CI Identifier	EVB_D_18306
Description	Describes the RU builder for an integrator who will put the builder into a full DAQ system
Point of Contact	S. Murray (Steven.Murray@cern.ch)
Physical Location	http://smurray.home.cern.ch/smurray/RUB_V3_9.doc

Table of Contents

1 Introduction 6

1.1 Document purpose and scope 6
1.2 Intended readership 6
1.3 System requirements and dependencies 7
1.4 References 7
1.5 Definitions, Acronyms and Abbreviations 7

2 RU builder overview 8

2.1 How the RU builder fits within the EVB 8
2.2 What the EVB does 9
2.3 The RU builder applications 9

3 RU builder application FIFOs 10

3.1 BU FIFOs 10
3.2 EVM FIFOs 11
3.3 RU FIFOs 12

4 I2O interface 13

4.1 TA/EVM interface 14
4.2 RU/RUI interface 16
4.3 BU/FU interface 17

5 Application state machines 21

5.1 Commonalities of the application finite state machines 21
5.2 BU, EVM and RU finite state machines 22

6 Starting the RU builder 23

7 Stopping the RU builder 24

8 Exported configuration parameters 25

9 How to obtain and build the RU builder 28

9.1 Checking out the source code from CVS 28
9.2 RU builder application and support libraries 29
9.3 Building the RU builder application and support libraries 29

10 Example configuration file 30

11 RU builder self test 31

12 Configuration guidelines 34

List of Figures

Figure 1 RU builder applications and how they interact with the rest of the EVB 8
Figure 2 BU FIFOs 10
Figure 3 EVM FIFOs.....11
Figure 4 RU FIFOs 12
Figure 5 External I2O interfaces of the RU builder.....13
Figure 6 TA/EVM interface sequence diagram.....14
Figure 7 RU/RUI interface sequence diagram.....16
Figure 8 BU/FU interface sequence diagram.....17
Figure 9 FSTN of a RU builder application.....21
Figure 10 BU, EVM and RU FSTNs.....22
Figure 11 HyperDAQ web page for self test 32
Figure 12 RUBuilderTester web page 32
Figure 13 RUBuilderTester control web page 33
Figure 14 Running RU builder 33

List of Tables

Table 1 Exported configuration parameters.....25

1 Introduction

The RU builder is a distributed XDAQ application that is part of a larger system called the event builder (EVB). The CMS data acquisition group is presently developing the EVB as described in the TriDAS TDR [1]. This document explains how to obtain, build and configure version 3.9 of the RU builder.

The new features introduced by this release are:

- The BU, EVM and RU now use the xdaq2rc library to send asynchronous state change messages to run-control
- The finite state machine (FSM) of the RU has been expanded to indicate whether or not the RU is tolerating or back-pressuring when it detects an event number mismatch or a time out in the event data stream
- The RU is now decoupled from BU instance numbers to facilitate the prototyping of multiple slices
- Bug fix – The BU and EVM now correctly resynchronize with the RU mask when they are configured
- The HTTP content type of the BU, EVM, RU and example applications is now explicitly set

This release was driven by the needs of central DAQ. However other users will be affected by the use of the xdaq2rc library and by the changes made to the FSM of the RU. The use of the xdaq2rc library is documented in chapters 9 and 10. The new finite state machine of the RU is documented in chapter 5.

1.1 Document purpose and scope

The goal of this document is to enable the reader to integrate the RU builder into a "running system" composed of the RU builder itself, a trigger source, one or more event data sources, one or more data sinks and some form of run-control. The RU builder cannot run without the components just listed. This document describes how to obtain, build and configure the RU builder. This document does not describe the other components of a "running system", such as run control software or how to setup a trigger or event data source. It is also not the purpose of this document to describe the internal workings of the RU builder. Developers are referred to the source code for such information. The code has been structured and commented so that it can be easily read and understood. It is recommended to use Doxygen to generate documentation from the code, as compatible comment tags have been used. If the reader is not familiar with Doxygen, then they are referred to its website: http://www.doxygen.org

It must be emphasized that the RU builder is still under development and subject to change. No description of the RU builder given in this document can be relied upon to be valid beyond this release.

1.2 Intended readership

This document is intended for a system integrator – someone that needs to integrate the RU builder into a data acquisition system (DAQ). It is assumed that the DAQ system is based on the XDAQ framework. If the reader is not familiar with this framework, then they are referred to the XDAQ website: http://xdaq.web.cern.ch/xdaq.

1.3 System requirements and dependencies

Version 3.9 of the RU builder depends on version 3.5 of the XDAQ core framework.

1.4 References

[1] The CMS collaboration, The Trigger and Data Acquisition project, Volume II, Data Acquisition & High-Level Trigger. CERN/LHCC 2002-26, ISBN 92-9083-111-4

1.5 Definitions, Acronyms and Abbreviations

BU	Builder Unit	***I2O***	Intelligent Input/Output
CVS	Concurrent Versioning System	***RU***	Readout Unit
DAQ	Data Acquisition system	***RUI***	Readout Unit Input
EVB	Event builder	***TA***	Trigger Adapter
EVM	Event Manager	***TDR***	Technical Design Report
FED	Front End Driver	***TriDAS***	Trigger and Data Acquisition
FSTN	Finite State Transition Network	***XDAQ***	Cross platform data acquisition toolkit
FU	Filter Unit		

2 RU builder overview

2.1 How the RU builder fits within the EVB

The RU builder is a component of a larger system called the event builder (EVB). The EVB is a distributed application that reads out event fragments from one set of nodes and assembles them into entire events in another set of nodes. Figure 1 shows the applications of the RU builder and how they interact with the rest of the EVB.

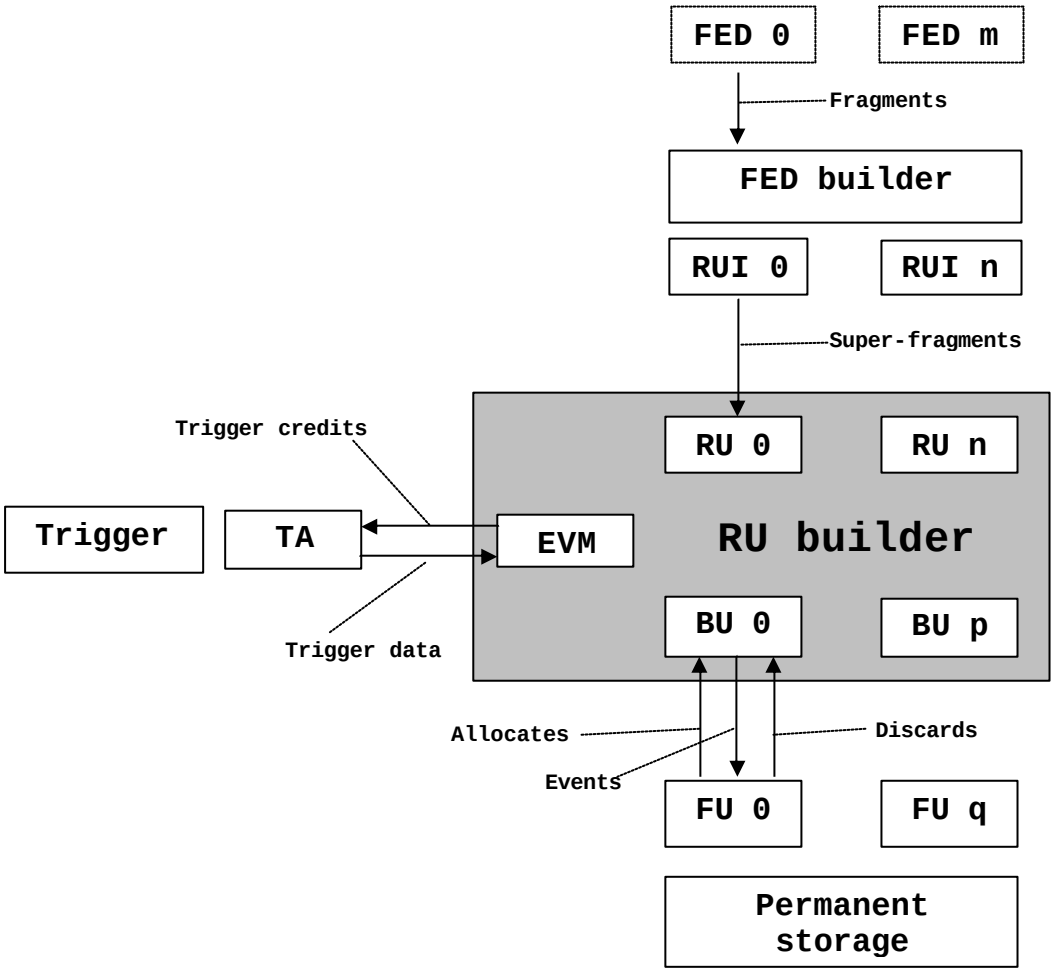

Figure 1 RU builder applications and how they interact with the rest of the EVB

The external interfaces of the RU builder assume that triggers are given to the EVM in the same order as their corresponding event data is given to the Rus.

2.2 What the EVB does

For each event the EVB:

- Reads out the trigger data. This trigger data will become the first super-fragment of the event.
- Reads out the fragments of the event from the detector front-end drivers (FEDs).
- Builds the fragments into RU super-fragments using the FED builder.
- Builds the whole event using the RU builder. The whole event is the trigger super-fragment plus the set of RU super-fragments.
- Decides whether or not the event is interesting for physics using the filter units (Fus).
- Sends the event to permanent storage if it is interesting for physics, or discards it if it is not.

2.3 The RU builder applications

The RU builder consists of a single event manager (EVM), one or more readout units (Rus) and one or more builder units (Bus). The EVM is responsible for controlling the flow of data through the RU builder. The Rus are responsible for buffering super-fragments until they are requested by the Bus. The Bus are responsible for building and buffering events until they are requested by the filter units (Fus).

The trigger adapter (TA), readout unit inputs (RUIs) and filter units (Fus) are external to the RU builder. The TA is responsible for interfacing the DAQ trigger to the EVM. The RUIs are responsible for pushing super-fragment data from the FED builder into the Rus. The Fus are responsible for selecting interesting events for permanent storage.

3 RU builder application FIFOs

The RU builder applications use FIFOs to keep track of requests, trigger data and event data. Knowledge of these FIFOs is required in order to correctly configure the RU builder. This chapter is divided into three sections, one for the BU, one for the EVM and one for the RU. Each section gives a brief description of the application's behavior and how its FIFOs are used.

3.1 BU FIFOs

A BU is responsible for building events. An event is composed of one trigger super-fragment and N RU super-fragments, where N is the number of Rus. To understand the internal FIFOs of a BU, it is first necessary to know its dynamic behavior. Figure 2 shows the internal FIFOs of a BU. With free capacity available, a BU requests the EVM to allocate it an event (***step 1***). The EVM confirms the allocation by sending the BU the event id and trigger data of an event (***step 2***). This trigger data is the first super-fragment of the event. The BU now requests the Rus to send it the rest of the event's super-fragments (***step 3***). The BU builds the super-fragments it receives from the Rus (***step 4***) into a whole event within its resource table (***step 5***). Fus can ask a BU to allocate them events (***step 6***). A BU services a FU request by asking the FU to take a whole event (***step 7***). When a FU has finished with an event, it tells the BU to discard it (***step 8***).

Each BU has its own worker thread that executes the behavior of that BU. The eventIdFIFO, blockFIFO, requestFIFOs and discardFIFO are used by the peer transport thread(s) to store incoming messages ready for the worker thread to process them. The `fullResourceFIFO` is manipulated solely by the worker thread. It is used to store which events were built in which order. This enables a BU to service a FU request with the next event that was built.

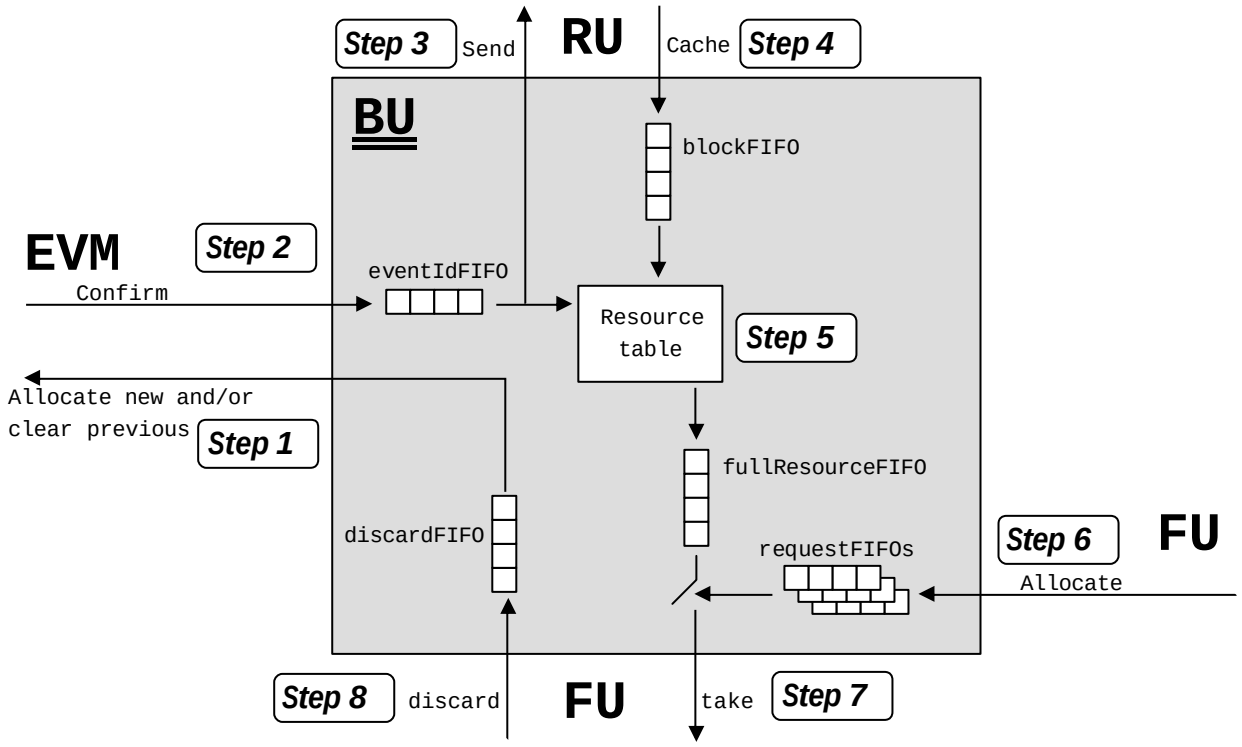

Figure 2 BU FIFOs

3.2 EVM FIFOs

The EVM is responsible for controlling the flow of event data through the RU builder. To understand the internal FIFOs of the EVM, it is first necessary to know its dynamic behavior. Figure 3 shows the internal FIFOs of the EVM. The EVM tells the TA the capacity of the RU builder by sending it trigger credits (***step 1***). One trigger credit represents the ability to build one event. Given a credit, the TA sends the EVM the trigger data of an event (***step 2***). The EVM pairs the trigger data with a free event id (***step 3***). The EVM also requests the Rus to readout the event's data (***step 4***). A BU with the ability to build an event will ask the EVM to allocate it an event (***step 5***). Within such a request, a BU will normally give back the id of an event to be cleared. For each cleared event id, the EVM sends a trigger credit to the TA and makes the id a free event id (***step 6***). The EVM confirms the allocation of an event by sending the requesting BU the event id and trigger data of the allocated event (***step 7***).

The EVM has a worker thread that executes the behavior of the EVM. The `triggerFIFO`, `clearedEventIdFIFO` and `requestFIFO` are used by the peer transport thread(s) to store incoming messages ready for the worker thread to process them. The `pairFIFO` and `freeEventIdFIFO` are manipulated solely by the worker thread. The `pairFIFO` keeps track of the "event id / trigger data" pairs that have yet to be sent to requesting Bus. The `freeEventIdFIFO` stores the ids of free events for which trigger credits have been sent to the TA.

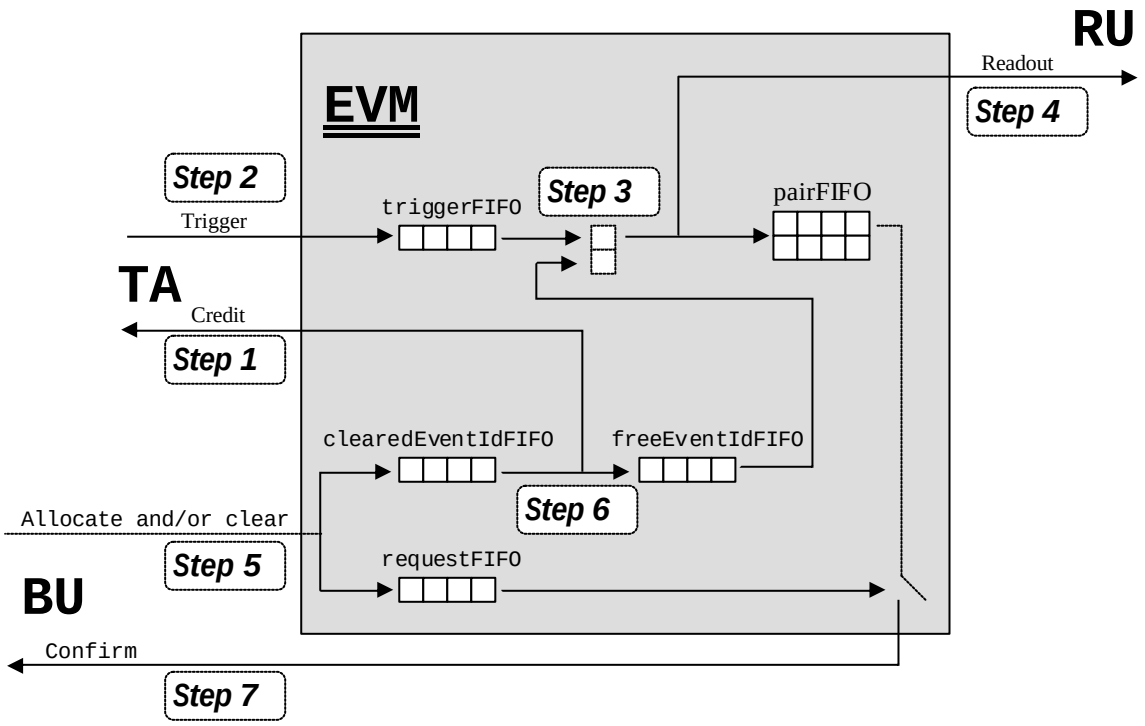

Figure 3 EVM FIFOs

3.3 RU FIFOs

A RU is responsible for buffering super-fragments until they are request by the Bus. To understand the internal FIFOs of a RU it is first necessary to know its dynamic behavior. Figure 4 shows the internal FIFOs of a RU. The EVM sends a RU an "event id / trigger event number" pair when it asks the RU to readout the corresponding event's data (***step 1***). In parallel, the RUI informs the RU of event data that is ready to be processed (***step 2***). A RU places each super-fragment for which it has received a pair into the fragment lookup table (***step 3***). Bus ask Rus to send them the super-fragments of the events they are building (***step 4***). A RU services a BU request by retrieving the super-fragment from its fragment lookup table and asking the BU to cache the super-fragment (***step 5***).

Each RU has a worker thread that executes the behavior of that RU. All of the internal FIFOs of a RU, that is to say the `pairFIFO`, `blockFIFO` and `requestFIFOs`, are used by the peer transport thread(s) to store incoming messages for the worker thread to process.

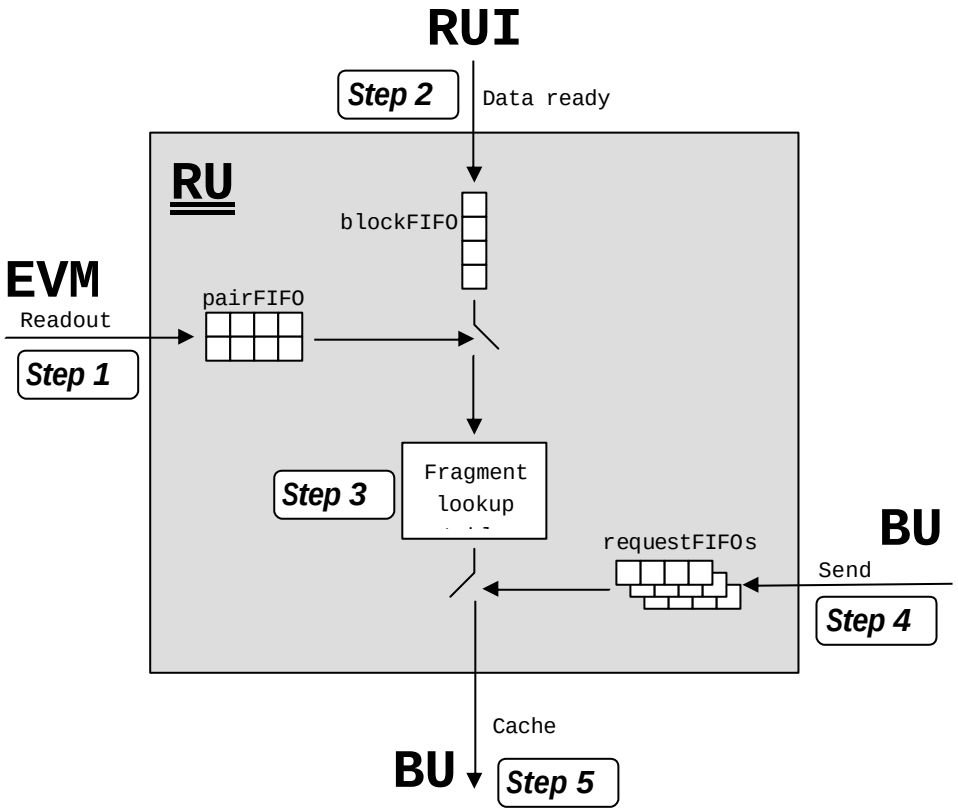

Figure 4 RU FIFOs

4 I2O interface

All the I2O messages of the EVB, including the internal and external messages of the RU builder, are defined in the package:

`TriDAS/daq/interface`

The I2O function codes of all the RU builder I2O messages are given in the file:

`TriDAS/daq/interface/shared/include/i2oXFunctionCodes.h`

The C structures that define the I2O messages are in the file:

`TriDAS/daq/interface/evb/include/i2oEVBMsgs.h`

The I2O interface of the RU builder is subject to change. The description of the interface provided by this document cannot be relied upon to be valid beyond this release.

Figure 5 shows the external I2O interfaces of the RU builder: the TA/EVM interface, the RUI/RU interface and the BU/FU interface. This chapter is divided into three sections, one for each interface.

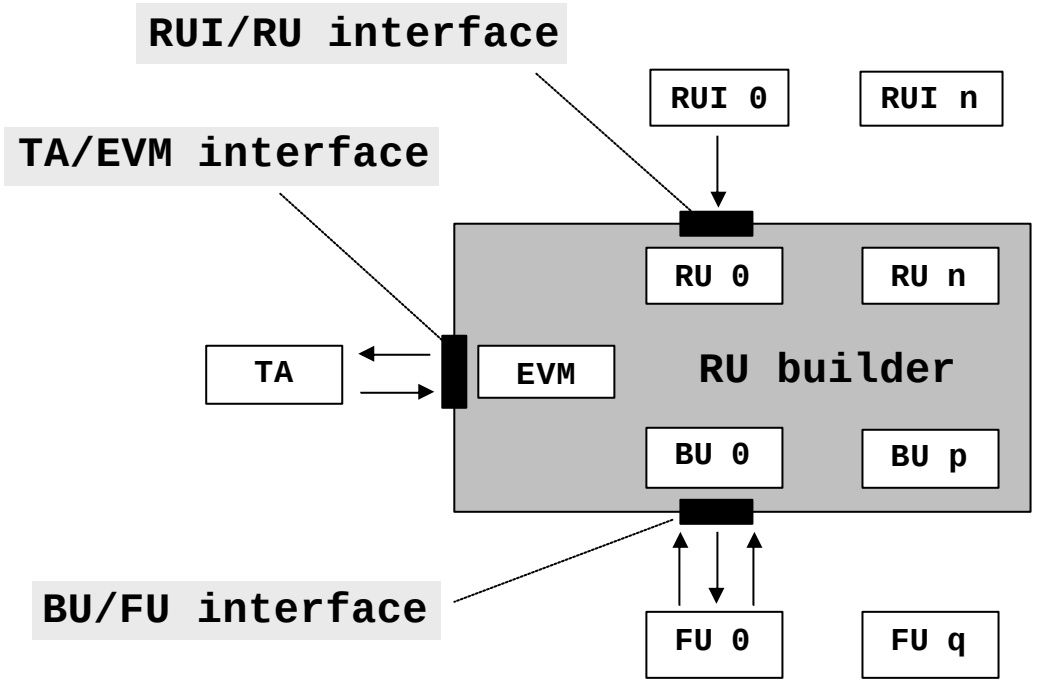

Figure 5 External I2O interfaces of the RU builder

4.1 TA/EVM interface

The TA/EVM interface specifies how:

- The EVM gives the TA trigger credits
- The TA gives the EVM trigger data

Figure 6 is a sequence diagram describing the protocol between the EVM and the TA.

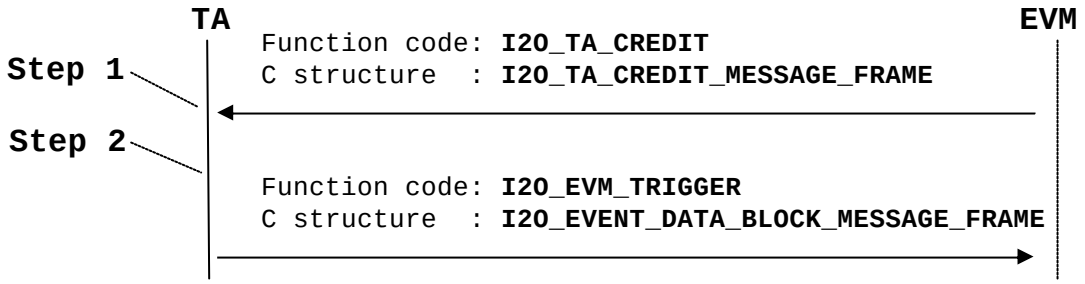

Figure 6 TA/EVM interface sequence diagram

The EVM communicates with the TA using a credit-based mechanism. The EVM tells the TA the current capacity of the RU builder by sending the TA a trigger credit count (***step 1***). One trigger credit represents the RU builder's ability to build one event. The TA should only send the EVM trigger data for as many events as the EVM has given the TA credits (***step 2***). The TA is responsible for getting / receiving trigger data from the trigger and for providing backpressure to the trigger as necessary.

The **I2O_TA_CREDIT_MESSAGE_FRAME** C structure is as follows:

```
typedef struct _I2O_TA_CREDIT_MESSAGE_FRAME
{
    I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
    U32 nbCredits;
} I2O_TA_CREDIT_MESSAGE_FRAME, *PI2O_TA_CREDIT_MESSAGE_FRAME;
```

The EVM must fill **nbCredits**.

The **I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME** C structure is as follows:

```
typedef struct I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME
{
    I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
    U32 eventNumber;
    U32 nbBlocksInSuperFragment;
    U32 blockNb;
    U32 eventId;
    U32 buResourceId;
    U32 fuTransactionId;
    U32 nbSuperFragmentsInEvent;
    U32 superFragmentNb;
    U32 padding;
} I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME, *PI2O_EVENT_DATA_BLOCK_MESSAGE_FRAME;
```

The TA must fill:

- **eventNumber**
- **nbBlocksInSuperFragment**
- **blockNb**

Version 3.0 of the RU builder only supports single block trigger data. Therefore the TA must set `nbBlocksInSuperFragment` to 1 and `blockNb` to 0

4.2 RU/RUI interface

The RU/RUI interface specifies how a RUI passes super-fragments to a RU. Figure 7 is a sequence diagram describing the protocol between the RUI and the RU.

Figure 7 RU/RUI interface sequence diagram

A super-fragment is composed of one or more **I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME**s. The **I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME** C structure is as follows:

```
typedef struct I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME
{
    I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
    U32 eventNumber;
    U32 nbBlocksInSuperFragment;
    U32 blockNb;
    U32 eventId;
    U32 buResourceId;
    U32 fuTransactionId;
    U32 nbSuperFragmentsInEvent;
    U32 superFragmentNb;
    U32 padding;
} I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME, *PI2O_EVENT_DATA_BLOCK_MESSAGE_FRAME;
```

The RUI must fill:

- **eventNumber**
- **nbBlocksInSuperFragment**
- **blockNb**

The **nbBlocksInSuperFragment** field gives the number of blocks the super-fragment is composed of. The **blockNb** field indicates the block's position within the super-fragment. Blocks are numbered from **0** to **nbBlocksInSuperFragment – 1**.

4.3 BU/FU interface

The BU/FU interface specifies how:

- A FU requests events from a BU
- A BU sends an event to a FU
- A FU tells a BU to discard an event

Figure 8 is a sequence diagram describing the protocol between a BU and a FU.

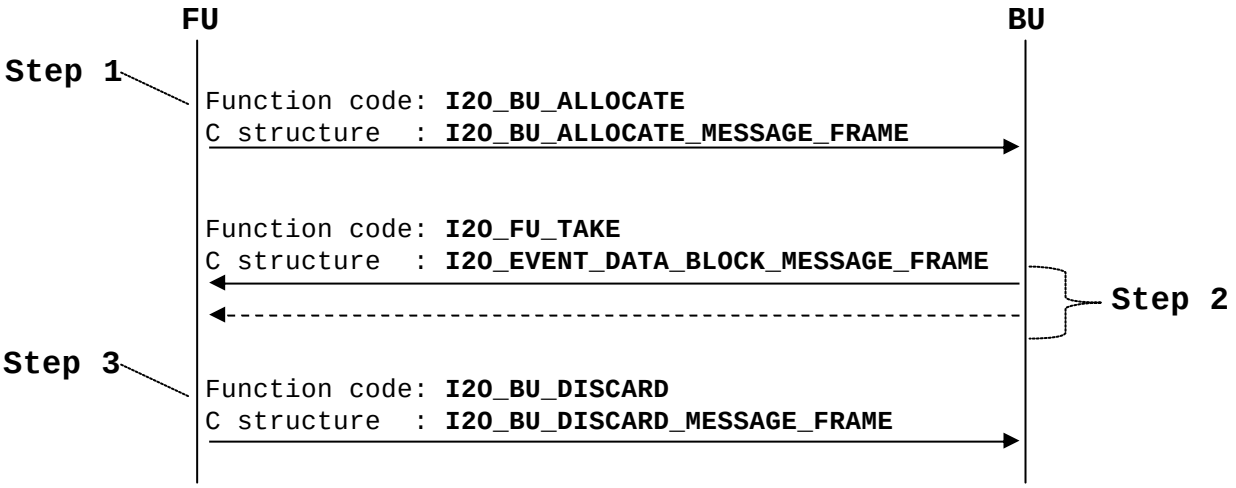

Figure 8 BU/FU interface sequence diagram

A FU requests a BU to allocate it one or more events (***step 1***). In response, the BU asks the FU to take the requested event data as a set of event data blocks (***step 2***). When a FU has finished processing one or more events, it tells the BU to discard them (***step 3***).

The BU/FU interface of this version of the RU builder does not support partial events. Partial events may be supported in a future version.

The **I2O_BU_ALLOCATE_MESSAGE_FRAME** C structure and its companion **BU_ALLOCATE** C structure are as follows:

```
typedef struct _BU_ALLOCATE
{
    U32 fuTransactionId;
    U32 fset;
} BU_ALLOCATE, *PBU_ALLOCATE;

typedef struct _I2O_BU_ALLOCATE_MESSAGE_FRAME {
    I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
    U32                       n;
    BU_ALLOCATE               allocate[1];
} I2O_BU_ALLOCATE_MESSAGE_FRAME, *PI2O_BU_ALLOCATE_MESSAGE_FRAME;
```

The FU must fill:

n
allocate[]

The **n** field specifies the number of events the FU is requesting. The **allocate** field is an array of FU transaction ids and fragment sets. For each event a FU requests, the FU fills in the **fuTransactionId** field and the **fset** field of a **BU_ALLOCATE** C structure and puts it in the **allocate** array. The **fuTransactionId** field is a transaction id that a FU can use to match its requests with the events it receives. A BU treats the **fuTransactionId** field as being opaque, in other words it is not interpreted. A BU will send back a copy of the **fuTransactionId** field in each of the **I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME**s that make up the requested event. The **fset** field is a fragment set identifier. Fragment sets are a way to describe partial events. The **fset** field is ignored by the BU in this version of the RU builder, because this version does not support partial events.

The **I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME** C structure is as follows:

```
typedef struct I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME
{
    I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
    U32 eventNumber;
    U32 nbBlocksInSuperFragment;
    U32 blockNb;
    U32 eventId;
    U32 buResourceId;
    U32 fuTransactionId;
    U32 nbSuperFragmentsInEvent;
    U32 superFragmentNb;
    U32 padding;
} I2O_EVENT_DATA_BLOCK_MESSAGE_FRAME, *PI2O_EVENT_DATA_BLOCK_MESSAGE_FRAME;
```

The FU should only read:

nbSuperFragmentsInEvent
superFragmentNb
nbBlocksInSuperFragment
blockNb
buResourceId
fuTransactionId

An event is composed of **I2O_EVENT_DATA_BLOCK_FRAME**s. The **nbSuperFragmentsInEvent**, **superFragmentNb**, **nbBlocksInSuperFragment**, **blockNb** fields are used to identify the position of an event data block within an event. An event is composed of one trigger super-fragment plus N RU super-fragments, where N is the number of Rus. Therefore the **nbSuperFragmentsInEvent** field is set to the number of Rus plus 1. The **superFragmentNb** field is numbered from **0** to **nbSuperFragmentInEvent – 1**. The **blockNb** field is numbered from **0** to **nbBlocksInSuperFragment – 1**.

The **buResourceId** field is an opaque handle that a FU should use to identify events/resources to be discarded. The **fuTransactionId** field is the FU transaction id of the FU request that caused the BU to reply with the current event.

The **I2O_BU_DISCARD** C structure is as follows:

```
typedef struct _I2O_BU_DISCARD_MESSAGE_FRAME {
   I2O_PRIVATE_MESSAGE_FRAME PvtMessageFrame;
   U32                       n;
   U32                       buResourceId[1];
} I2O_BU_DISCARD_MESSAGE_FRAME, *PI2O_BU_DISCARD_MESSAGE_FRAME;
```

The FU must fill:

- **n**
- **buResourceId[]**

The **n** field specifies the number of events/resources to be discarded. The **buResourceId** field is an array of the ids of the BU resources to be discarded.

5 Application state machines

5.1 Commonalities of the application finite state machines

The finite state machines of the Bus, EVM and Rus have commonalities. Figure 9 shows the finite state transition network (FSTN) which all three types of application follow. There are four common behaviors. Firstly, all RU builder applications read and act upon configuration parameters when they receive a Configure SOAP message. Secondly, all RU builder applications only participate in event building when they are enabled. Thirdly, all RU builder applications throw away their internal data and any incoming I2O message frames when they are halted. Fourthly all RU builder applications go to the Failed state when they encounter a fatal error. A RU builder application cannot be recovered from the Failed state.

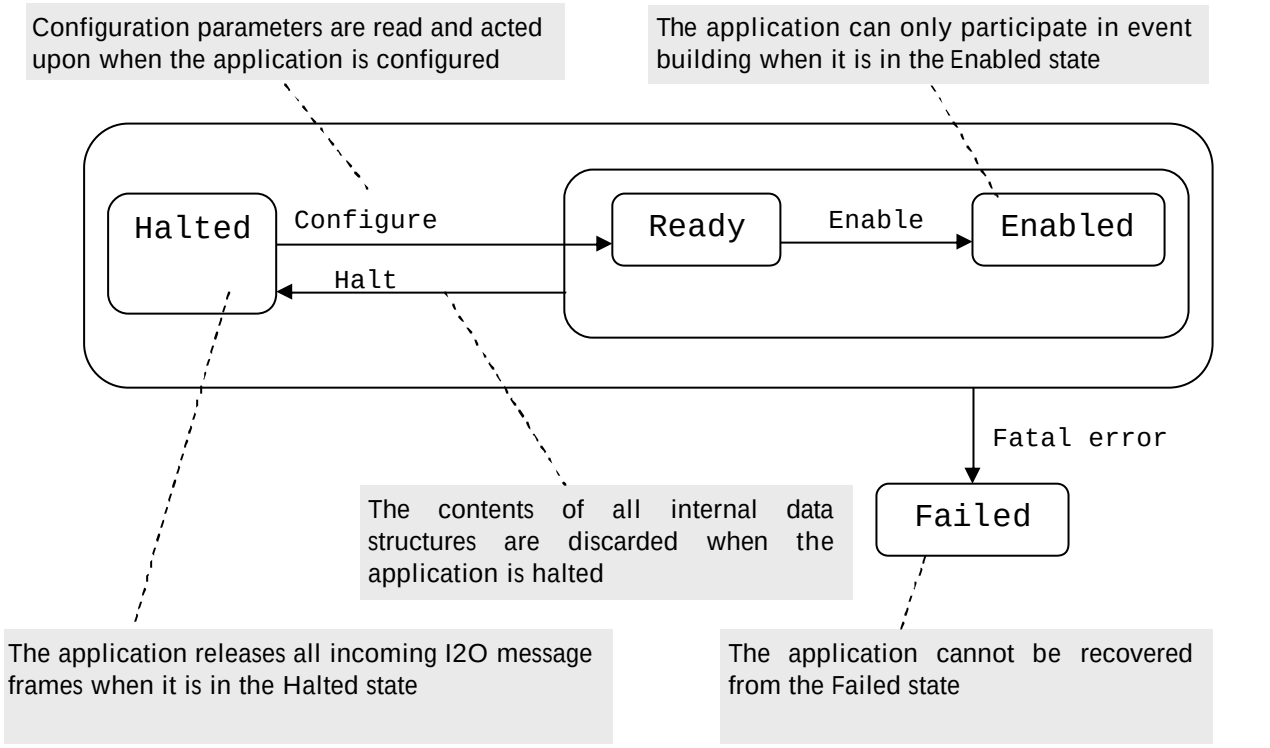
Configuration parameters are read and acted upon when the application is configured
The application can only participate in event building when it is in the Enabled state
Halted
Configure
Ready
Enable
Enabled
Halt
Fatal error
Failed
The contents of all internal data structures are discarded when the application is halted
The application releases all incoming I2O message frames when it is in the Halted state
The application cannot be recovered from the Failed state

Figure 9 FSTN of a RU builder application

5.2 BU, EVM and RU finite state machines

The FSTNs specific to each type of RU builder application are shown in figure 10.

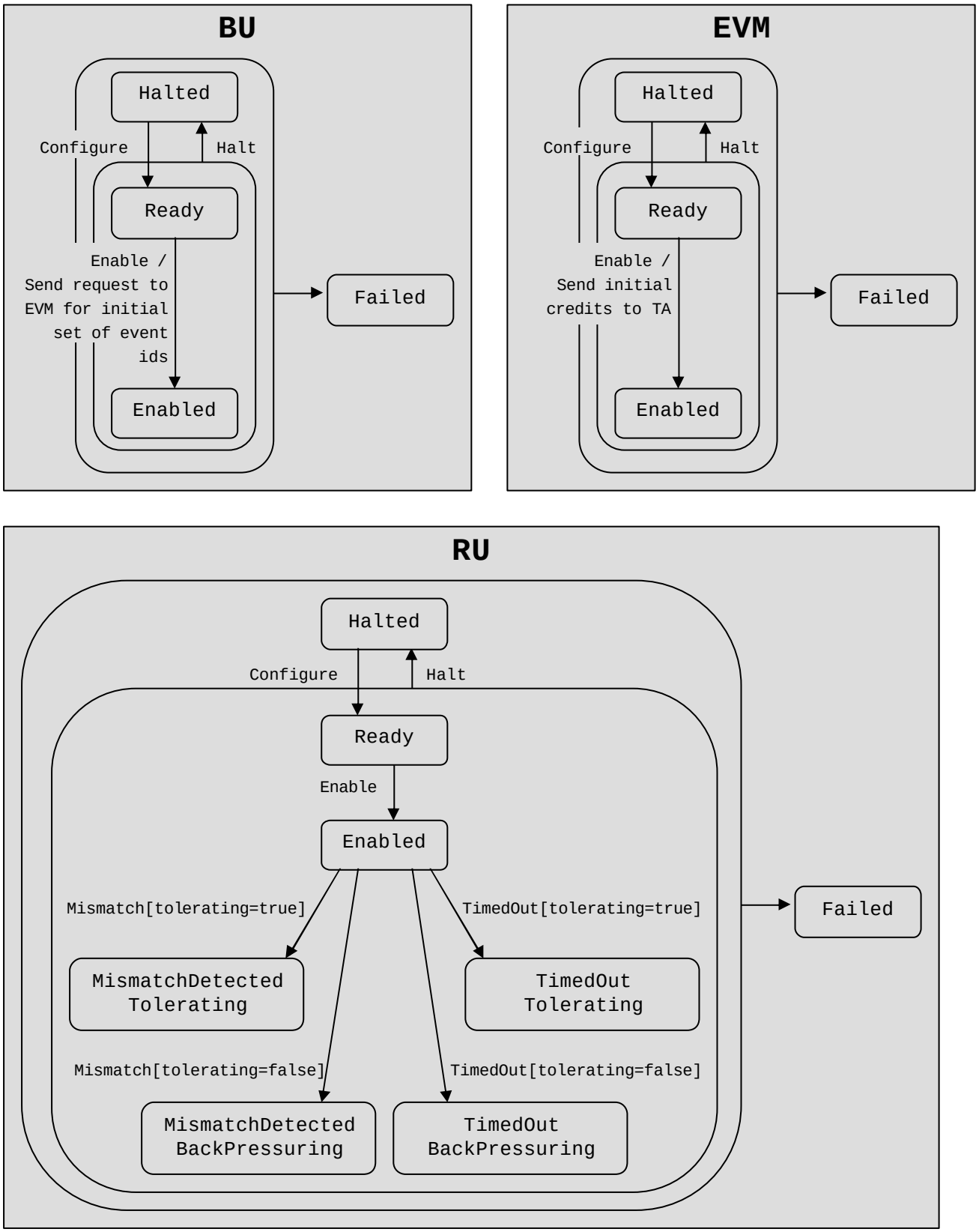

Figure 10 BU, EVM and RU FSTNs

6 Starting the RU builder

The RU builder is part of a larger system called the event builder (EVB). Besides run-control, the RU builder communicates with a TA, one or more RUIs and one of more Fus. The RU builder cannot be started at any arbitrary moment in time; its start-up must be synchronized with that of the TA, RUIs and Fus. The RU builder and the EVB components it interacts with are XDAQ applications, and as such depend on one or more peer transports to communicate with each other. These peer transport must be up and running before these applications try to communicate with each other. To start the RU builder and the components it interacts with, run-control should do the following in order:

1. Start the necessary peer transports so that the RU builder and its surrounding applications can communicate
2. Start the TA so that is can service credits from the EVM
3. Start the RU builder so that it can receive super-fragments from the RUIs
4. Start the RUIs and Fus so they can start pushing in super-fragments and extracting events respectively

The RU builder is a distributed application whose nodes (Bus, EVM and Rus) need to be started up in a specific order. To put the RU builder into the state where it will build events, run-control should do the following in order:

1. Send Configure to all of the RU builder applications
2. Send Enable to all of the Rus
3. Send Enable to the EVM
4. Send Enable to the Bus

Unlike version 2.x of the RU builder, the state changes of this version are synchronous. There is no need to poll the state of an application after a request to change state.

The Rus have to be enabled first because they have to be ready to receive “event number” / “event id” pairs from the EVM. The EVM can start sending these pairs immediately after it has been enable.

The EVM has to be enabled before the Bus so that it is ready to service their requests for event ids. Bus can start requesting event ids as soon as they are enabled. Enabling the EVM causes it to send an initial trigger credit count to the TA. The number of initial credits is equal to the total number of event ids in the RU builder. As soon as trigger data arrives at the EVM from the TA, the EVM sends “event number” / “event id” pairs to the Rus. As explained in the previous paragraph, this is why the Rus have to be enabled before the EVM.

Enabling a BU causes it to send its initial request for event ids to the EVM. The number of initial event ids requested is equal to the maximum number of event ids the BU is allowed to acquire at any single moment in time.

7 Stopping the RU builder

The current version of RU builder foresees two ways of stopping the RU builder:

- Stop the trigger and event data entering the RU builder
- Halt all of the RU builder applications

When stopping the trigger and event data entering the RU builder, it is useful to know when the RU builder has finished building the events for which it has received triggers and event data. This can be found out by reading the following exported parameter of the EVM:

```
EVM::ruBuilderIsEmpty
```

Halting a RU builder application causes it to discard (destroy) all the data in its internal data structures and to release all incoming I2O messages.

8 Exported configuration parameters

Configuration parameters need to be set before an application is sent a Configure SOAP message.

Table 1 lists the exported control parameters of each type of RU builder application. The type and default value of each parameter is given.

APP	*PARAMETER NAME*	*TYPE*	*VALUE*
BU, EVM & RU	nbEvtIdsInBuilder	unsigned long	4096
BU, EVM & RU	ageMessages	bool	true
BU, EVM & RU	msgAgeLimitDtMSec	unsigned long	1000
BU, EVM & RU	exitOnFail	bool	false
BU	blockFIFOCapacity	unsigned long	16384
BU	discardFIFOCapacity	unsigned long	65536
BU	I2O_EVM_ALLOCATE_CLEAR_Packing	unsigned long	8
BU	maxEvtsUnderConstruction	unsigned long	64
BU	requestFIFOCapacity	unsigned long	1024
BU	I2O_RU_SEND_Packing	unsigned long	8
EVM	sendCreditsWithDispatchFrame	bool	false
EVM	triggerFIFOCapacity	unsigned long	4096
EVM	I2O_RU_READOUT_Packing	unsigned long	8
EVM	I2O_TA_CREDIT_Packing	unsigned long	8
RU	blockFIFOCapacity	unsigned long	16384

Table 1 Exported configuration parameters

The default values are set when the RU builder application is instantiated. The default values have been chosen with the goal of covering the majority of use-cases for the RU builder. A user should rarely need to diverge from these default values.

The following assumptions were made when calculating the default values of the RU builder's configuration parameters:

- A RU builder is composed of 64 BUs and 64 RUs.
- A RU has 64MB of physical memory for caching super-fragments
- An event is 1MB
- An event is made up of 64 super-fragments (1 per RU) of equal size; therefore the size of a super-fragment is 16KB.
- The block size (size of an I2O message frame used to transport event data) is 4KB
- The RUIs only give as many events to the RUs as the TA gives triggers to the EVM
- The maximum number of FUs per BU is 64
- A FU will never have more than 1024 outstanding requests for events
- Fast control messages are sent if they are older than 1 second
- The packing factor of fast control messages is 8

The need to know the total number of event ids in the RU builder is common to all three types of RU builder applications.

- The total number of event ids in the RU builder is a function of RU memory. Assuming each RU has 64MB of memory for buffering super-fragments and that the size of an event is 1 MB:

  `BU, EVM & RU::nbEvtIdsInBuilder` = sum of the memory of all RUs / size of an event
  = (64 × 64MB) / 1MB
  = 4096

The BUs, and EVM send fast control messages.

- Fast control messages are sent if they are older than 1 second

  `BU & EVM::ageMessages` = true
  `BU & EVM::msgAgeLimitDtMSec` = 1000

It has been assumed that all events have the fixed size of 1MB. If the RU builder is to build events of varying sizes then the appropriate safety factor needs to be taken into account when calculating the number of event ids in the RU builder.

The default values of the BU control parameters were calculated as follows:

- To prevent a BU from monopolizing event ids, each BU has a maximum number of event ids it can acquire at any moment in time. Assuming all BUs are equal, each BU is allowed to acquire:

  BU::maxEvtsUnderConstruction = **nbEvtIdsInBuilder** / number of BUs
  = 4096 / 64
  = 64

- The blockFIFO of a BU (see figure 2 in section 3.2) is responsible for buffering incoming event data. In the worst case this FIFO would have to buffer the blocks of all outstanding requests for event data:

  BU::blockFIFOCapacity = **maxEvtIdsUnderConstruction** ×
  size of an event / block size
  = 64 × 1 MB / 4 KB
  = 64 ×256
  = 16384

- A BU has a single FIFO called discardFIFO for FU discard messages, and one FIFO per FU called requestFIFO for FU request messages. Knowing that a BU can service a maximum of 64 FUs and assuming that a single FU will never have more than 1024 outstanding requests for events:

  BU::discardFIFOCapacity = 1024 × maximum number of FUs
  = 1024 × 64
  = 65536

  BU::requestFIFOCapacity = 1024

The default values of the RU exported parameters were calculated as follows:

- The blockFIFO of the RU is responsible for buffering incoming super-fragment data. Assuming a RUI only gives as many super-fragments to a RU as the TA gives triggers to the EVM, then in the worst case the blockFIFO must hold the blocks of as many super-fragments as there are event ids in the RU builder:

  blockFIFOCapacity = **nbEvtIdsInBuilder** × (size of a super-fragment / block size)
  = 4096 × (16K / 4KB)
  = 16384

The parameters of the form **_Packing** should not normally be modified as they have only been tested with the default value of 8. However the user may modify them if they are experiencing performance problems with the RU builder.

9 How to obtain and build the RU builder

This chapter is divided into two sections. The first explains how to obtain the RU builder source code and the second explains how to build it. Both sections assume the following:

- The user has already installed the XDAQ core framework
- The user has defined the shell environment variable **$XDAQ_ROOT** to point to the root of their XDAQ installation, in other words their TriDAS directory
- The shell of the user is **tsch**
- The user accesses the CMS CVS server as an anonymous read-only user

9.1 Checking out the source code from CVS

The RU builder source code is stored in the CMS CVS server. For more information about this server, please look at the webpage http://cmsdoc.cern.ch/cmsoo/projects/cvs_server.html. You must login to the CMS CVS server if you wish to use it. To do so, enter the following:

```
setenv CVSROOT :pserver:anonymous@cmscvs.cern.ch:/cvs_server/repositories/TriDAS
cvs login
```

You will be prompted for a password:

```
CVS password:
```

Please enter:

```
98passwd
```

The RU builder source code needs to be placed in the directory **$XDAQ_ROOT/daq/evb**. Enter the following commands to put it there using the CMS CVS server:

```
cd $XDAQ_ROOT/..
cvs export -r EVB_S_18305_V3_9 TriDAS/daq/evb
```

The RU builder source code depends on the interface directory **$XDAQ_ROOT/daq/interface**. Enter the following commands to obtain this directory from the CMS CVS server.

```
cd $XDAQ_ROOT/..
cvs export –r EVB_S_18305_V3_9 TriDAS/daq/interface
```

The RU builder source code depends on the xdaq2rc library directory **$XDAQ_ROOT/RunControl/tools/xdaq2rc**. Enter the following commands to obtain this directory from the CMS CVS server.

```
cd $XDAQ_ROOT/..
cvs export –r EVB_S_18305_V3_9 TriDAS/RunControl/tools/xdaq2rc
```

9.2 RU builder application and support libraries

The RU builder is composed of seven application libraries:

```
TriDAS/daq/evb/bu/lib/linux/x86/libBU.so
TriDAS/daq/evb/evm/lib/linux/x86/libEVM.so
TriDAS/daq/evb/ru/lib/linux/x86/libRU.so

TriDAS/daq/evb/examples/fu/lib/linux/x86/libFU.so
TriDAS/daq/evb/examples/rui/lib/linux/x86/libRUI.so
TriDAS/daq/evb/examples/ta/lib/linux/x86/libTA.so

TriDAS/daq/evb/rubuildertester/lib/linux/x86/libRUBuilderTester.so
```

These libraries use the xdaq2rc library to send messages from XDAQ to run-control:

```
TriDAS/RunControl/tools/xdaq2rc/lib/linux/x86/libRUBuilderTester.so
```

In other words:

libBU.so depends on **libxdaq2rc.so**
libEVM.so depends on **libxdaq2rc.so**
libRU.so depends on **libxdaq2rc.so**

See chapter 10 for how these dependencies effect the XDAQ configuration of a RU builder.

9.3 Building the RU builder application and support libraries

Enter the following to build the xdaq2rc library and the RU builder application libraries:

```
cd $XDAQ_ROOT/ RunControl/tools/xdaq2rc
make
cd $XDAQ_ROOT/daq/evb
make
```

10Example configuration file

The following example configuration file shows how to configure a 1x1 RU builder to run on a single XDAQ executive:

```
<xc:Partition xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xc="http://xdaq.web.cern.ch/xdaq/xsd/2004/XMLConfiguration-30"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<i2o:protocol xmlns:i2o="http://xdaq.web.cern.ch/xdaq/xsd/2004/I2OConfiguration-30">
  <i2o:target class="EVM" instance="0" tid="23"/>
  <i2o:target class="RU"  instance="0" tid="24"/>
  <i2o:target class="BU"  instance="0" tid="25"/>
</i2o:protocol>

<xc:Context url="http://localhost:65432">
  <xc:Module>$XDAQ_ROOT/RunControl/tools/xdaq2rc/lib/linux/x86/libxdaq2rc.so</xc:Module>

  <xc:Application class="RUBuilderTester" id="12" instance="0" network="local"/>
  <xc:Module>$XDAQ_ROOT/daq/evb/rubuildertester/lib/linux/x86/libRUBuilderTester.so</xc:Module>

  <xc:Application class="EVM" id="13" instance="0" network="local"/>
  <xc:Module>$XDAQ_ROOT/daq/evb/evm/lib/linux/x86/libEVM.so</xc:Module>

  <xc:Application class="RU" id="14" instance="0" network="local"/>
  <xc:Module>$XDAQ_ROOT/daq/evb/ru/lib/linux/x86/libRU.so</xc:Module>

  <xc:Application class="BU" id="15" instance="0" network="local"/>
  <xc:Module>$XDAQ_ROOT/daq/evb/bu/lib/linux/x86/libBU.so</xc:Module>
</xc:Context>

</xc:Partition>
```

Two import rules for the module tag

1. The module tag for libxdaq2rc.so must appear once within each context tag where there is a module tag for libBU.so, libEVM.so or libRU.so.
2. The module tag for libxdaq2rc.so must appear within the context tag before any of the module tags libBU.so, libEVM.so or libRU.so

11 RU builder self test

This section explains how to perform the self test of the RU builder. This test helps determine whether or not the RU builder has been successfully installed.

The self test consists of the RUBuilderTester application plus one EVM, one RU and one BU all running on the same XDAQ executive. The EVM is told to generate dummy triggers, the RU is told to generate dummy super-fragments and the BU is told to drop the events it builds The step by step instructions to run the self test are:

Step 1
Obtain and build the RU builder. See section 9 for instructions.

Step 2
Open a terminal and create the XML configuration file for the self test by typing:

```
cd $XDAQ_ROOT/daq/evb/xml
./produceSelfTestXml.pl HOST PORT $XDAQ_ROOT 1x1SingleXDAQ.template.xml > 1x1SingleXDAQ.xml
```

Where ***HOST*** and ***PORT*** indicate the port on which the executive shall service HTTP requests.

Step 3
Run the XDAQ executive by typing :

```
$XDAQ_ROOT/daq/xdaq/bin/linux/x86/xdaq.sh –h HOST –p PORT –c 1x1SingleXDAQ.xml
```

Where ***HOST*** and ***PORT*** are the same as those given in step 2.

Step 4

Open a web browser and enter the following URL

`http://HOST:PORT`

Where ***HOST*** and ***PORT*** are the same as those of steps 2 and 3.

You should now see the HyperDAQ page of the XDAQ executive you started in step 3. It should look similar to the following screenshot.

Figure 11 HyperDAQ web page for self test

Step 5

Go to the default web page of the RUBuilderTester application by clicking on the “RUBuilderTester” link. Your web browser should now display something similar to:

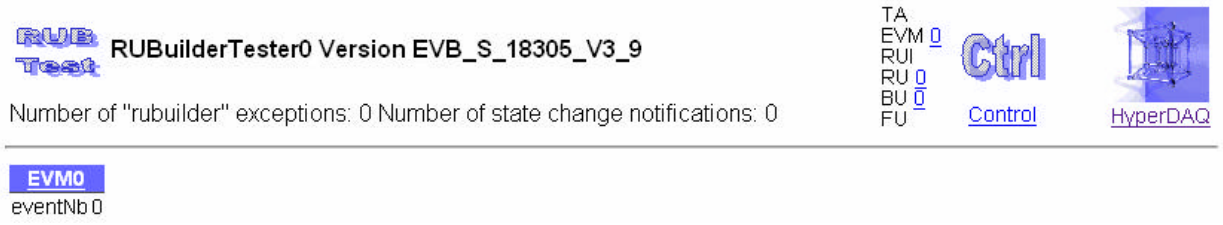

Figure 12 RUBuilderTester web page

Step 6

Go to the control web page by clicking on the “Ctrl” icon. You should now see something like:

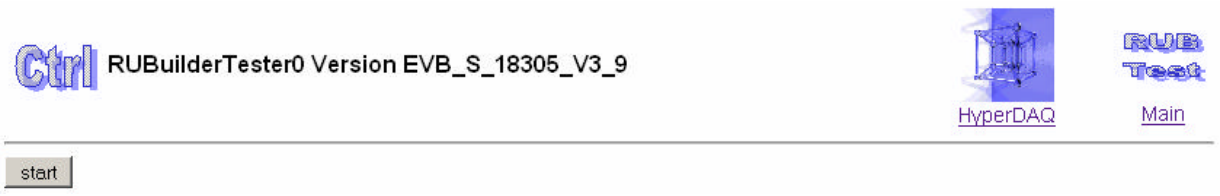

Figure 13 RUBuilderTester control web page

Step 7

Start the self test by clicking the “start” button. The button should now be labeled “stop”.

Step 8

Return to the main web page of the RU builder tester by clicking on the “RUB Test” icon. Clicking the refresh button of your browser should now show events being built. For example the “eventNb” variable of the EVM0 should increment.

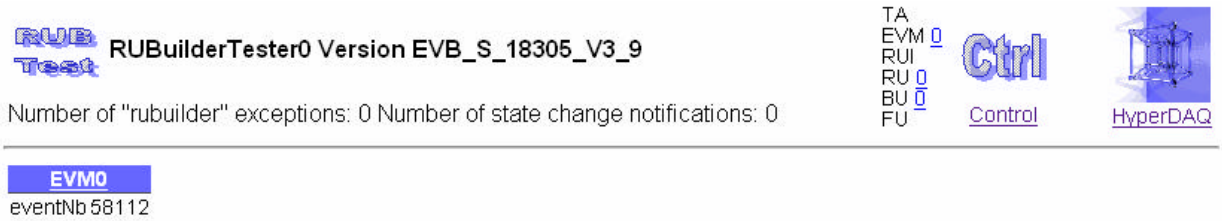

Figure 14 Running RU builder

12 Configuration guidelines

This chapter summarizes and highlights the most important points with regards to configuring the RU builder.

The default values of the RU builder control parameters have been chosen for the majority of use-cases. The user should rarely need to diverge from these values.

The RU builder is dependent on the instance numbers of the BUs, EVM, FUs, RUs, and TA:

- RUs must be assigned instance numbers from 0 to the number of RUs – 1
- BUs must be assigned instance number from 0 to the number of BUs – 1
- The EVM must be assigned instance number 0
- The TA must be assigned instance number 0

The RU builder has the following configuration restrictions:

- A single BU can service a maximum of 64 FUs.
- The sum of the maximum number of event ids each BU can have at any moment in time (**`BU::maxEvtsUnderConstruction`**) must not exceed the total number of event ids in the RU builder (**`BU, EVM, RU::nbEvtIdsInBuilder`**). If this is the case, then there is no guarantee that the EVM will be able to buffer BU requests for event ids, or that the RUs will be able to buffer BU requests for event data.
- The configuration parameters of a RU builder application must be set before it is configured.